National Research Council Canada
Institute for Information Technology

Conseil national de recherches Canada
Institut de Technologie de l'information

ERB-1088

NRC·CNRC

ES2: A Tool for Collecting Object-Oriented Design Metrics from C++ and Java Source Code

Stojanovic, M. and El Emam, K.

June 2001

Canada

NRC 44888

National Research Council Canada
Conseil national de recherches Canada

Institute for Information Technology
Institut de Technologie de l'information

ES2: A Tool for Collecting Object-Oriented Design Metrics from C++ and Java Source Code

Stojanovic, M. and El Emam, K.

June 2001

Copyright 2001 by
National Research Council of Canada

Permission is granted to quote short excerpts and to reproduce figures and tables from this report, provided that the source of such material is fully acknowledged.

105-04929/

ES2: A Tool for Collecting Object-Oriented Design Metrics from C++ and Java Source Code

Marta Stojanovic **Khaled El Emam**
National Research Council of Canada
Institute for Information Technology
Building M-50, Montreal Road
Ottawa, Ontario
Canada K1A OR6
{marta.stojanovic, khaled.el-emam}@nrc.ca

Abstract

There is considerable evidence that object-oriented design metrics can be used to make quality management decisions leading to substantial cost savings. However, one of the major impediments to the wider adoption of object-oriented metrics in practice has been the unavailability of robust and low-cost metrics analyzers. Commercial tools tend to collect basic size metrics and many variants of these size metrics. Other potentially powerful coupling metrics are typically not collected. This report is the user manual for a publicly available open-source metrics analyzer for the C++ and Java languages. This analyzer collects a set of design metrics that have been empirically demonstrated to be beneficial for making quality management decisions in practice.

1 Introduction

Recent evidence indicates that most faults in software applications are found in only a few of a system's components [6, 9-11]. The early identification of these components allows an organization to take mitigating actions, such as focus defect detection activities on high risk components, for example by optimally allocating testing resources [8], or redesigning components that are likely to cause field failures.

In the realm of object-oriented systems, one approach to identify faulty classes early in development is to construct prediction models using object-oriented design metrics. Such models are developed using historical data, and can then be applied for identifying potentially faulty-classes in future applications or future releases. The usage of design metrics allows the organization to take mitigating actions early and consequently avoid costly rework. Recent evidence suggests that taking such actions (in this case design inspections) can result in an estimated 42% saving in post-release costs for a Java application [7].

One of the difficulties in implementing quality management using object-oriented design metrics is that there have not been too many good tools available. A number of commercial tools collect the CK metrics suite [1] (for the tools, see [2] and also visit www.object-oriented.org). However, doubt has been raised about the validity of some of these metrics and about the methodology used in previous studies that

validated these metrics [5]. Therefore, there is a need for tools that implement other metrics that have been empirically validated in a methodologically sound manner.

This document describes a tool that can be used for collecting interface object-oriented metrics. This means that the metrics can be calculated from interface specifications, which are usually available at the design stage of a project. The current version of the tool works with the C++ and Java language. For C++ it can compute the metrics from the ".h" files (which are, in fact, the interface specifications). For Java, the ".java" files are used, but they can contain only method signatures, and not their actual implementation, in order to calculate the metrics (which enables us again to calculate metrics in the design phase). The metrics that are actually collected capture size, coupling, and inheritance. We only collect these metrics because previous research has indicated that they are good predictors of class fault-proneness [2-4, 7]. The metrics that are collected by the tool are summarized in Table 1.

Before proceeding further, it is important to realize that we are focusing only on metrics that were properly validated. That is why the tool does not collect all or many metrics that have been defined in the literature. As evidence accumulates demonstrating the efficacy of other metrics, they will certainly be included within our tool.

Title	Description
Coupling Metrics	
OCAIC (instances)	Class-attribute import coupling (number of coupled attributes) [1]
OCAIC (classes)	Class-attribute import coupling (number of coupled classes) [1]
OCAEC (instances)	Class-attribute export coupling (number of coupled attributes) [1]
OCAEC (classes)	Class-attribute export coupling (number of coupled classes) [1]
OCMIC (instances)	Class-method import coupling (number of coupled method parameters and return types)
OCMIC (classes)	Class-method import coupling (number of coupled classes)
OCMEC (instances)	Class-method export coupling (number of coupled method parameters and return types)
OCMEC (classes)	Class-method export coupling (number of coupled classes)
ACAIC (instances)	Ancestor class-attribute import coupling (number of coupled attributes) [1]
ACAIC (classes)	Ancestor class-attribute import coupling (number of coupled classes) [1]
DCAEC (instances)	Descendant class-attribute export coupling (number of coupled attributes) [1]
DCAEC (classes)	Descendant class-attribute export coupling (number of coupled classes) [1]
ACMIC (instances)	Ancestor class-method import coupling (number of coupled method parameters and return types)
ACMIC (classes)	Ancestor class-method import coupling (number of coupled classes)
DCMEC (instances)	Descendant class-method export coupling (number of coupled method parameters and return types)
DCMEC (classes)	Descendant class-method export coupling (number of coupled classes)
Inheritance Metrics	
DIT	Depth of inheritance tree
Size Metrics	
Number of pub/priv/ prot/all attributes	Number of attributes (depending on the access option: public, private, protected, all)
Number of pub/priv/ prot/all methods	Number of methods (depending on the access option: public, private, protected, all)
LOC	Lines of code (physical)

Table 1: Metrics collected by the ES2 tool.

1.1 Examples of Calculating the Metrics

Just to illustrate what the metrics mean, in this subsection we describe, using an example, a simple calculation of all of these metrics for C++ soource code. From Table 1 we can see that there are three types of interactions between 2 classes: their relationship (whether it is a parent-child relationship or other), type of interaction (whether a class has an attribute, method parameter, or return type of another class) and locus of impact (whether a class is using another class through its attributes or method types –

[1] Only defined for the C++ code, see section 4.2.2

import – or it is used by another class - export). Letters in the acronyms stand for these 3 types of interactions: A for ancestor, D for descendant, O for other (neither ancestor nor descendant), CA for class-attribute, CM for class-method, IC for import coupling and EC for export coupling.

Figure 1 shows a simple example of C++ code and how every type of coupling is captured.

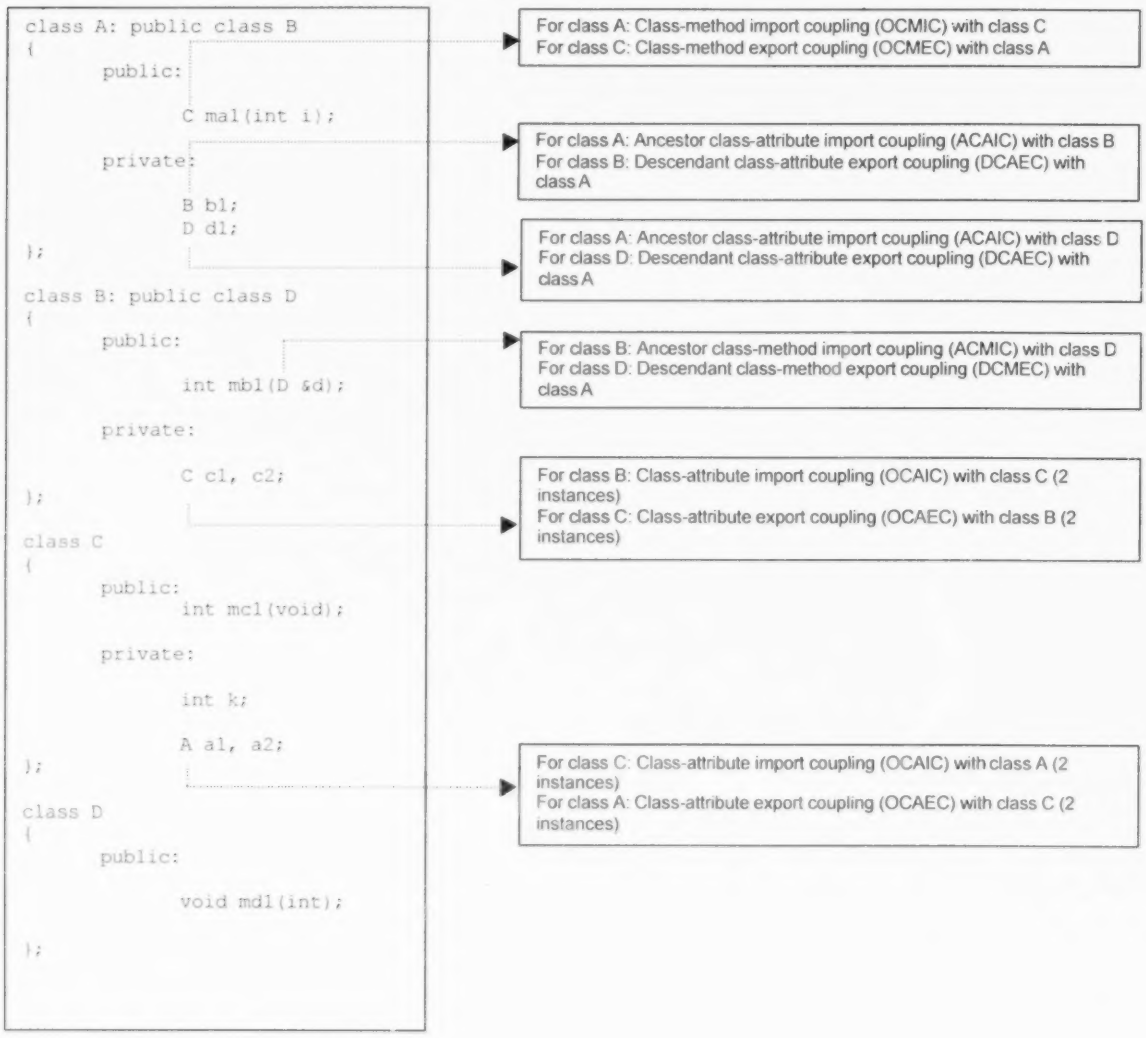
class A: public class B
{
public:
C ma1(int i);
private:
B b1;
D d1;
};
class B: public class D
{
public:
int mb1(D &d);
private:
C c1, c2;
};
class C
{
public:
int mc1(void);
private:
int k;
A a1, a2;
};
class D
{
public:
void md1(int);
};
For class A: Class-method import coupling (OCMIC) with class C
For class C: Class-method export coupling (OCMEC) with class A
For class A: Ancestor class-attribute import coupling (ACAIC) with class B
For class B: Descendant class-attribute export coupling (DCAEC) with class A
For class A: Ancestor class-attribute import coupling (ACAIC) with class D
For class D: Descendant class-attribute export coupling (DCAEC) with class A
For class B: Ancestor class-method import coupling (ACMIC) with class D
For class D: Descendant class-method export coupling (DCMEC) with class A
For class B: Class-attribute import coupling (OCAIC) with class C (2 instances)
For class C: Class-attribute export coupling (OCAEC) with class B (2 instances)
For class C: Class-attribute import coupling (OCAIC) with class A (2 instances)
For class A: Class-attribute export coupling (OCAEC) with class C (2 instances)

Figure 1: A simple C++ example for calculating metrics. This type of class definition code is typically in the interface ".h" files.

If we consider all the access types (public and private, in this case), we can see that class B, for example, has two attributes which are both of type C. If we count coupling instances, then it means that OCAIC(B)=2. Class B has one method whose argument type is of type D. As class D is a parent class of class B, we are talking about ancestor coupling, i.e., ACMIC(B) = 1. Class B has only one parent, class D, so its DIT is 1 (the DIT of class A is 2, because it is a subclass of class B, which is a subclass of class D). Similarly we can calculate all the metrics for all the classes. At the end, we would obtain the results presented in Table 2.

Metric	**Class A**	**Class B**	**Class C**	**Class D**	**Class A**	**Class B**	**Class C**	**Class D**
OCAIC (instances)	0	2	2	0	0	0	0	0
OCAIC (classes)	0	1	1	0	0	0	0	0
OCAEC (instances)	2	0	2	0	0	0	0	0
OCAEC (classes)	1	0	1	0	0	0	0	0
OCMIC (instances)	1	0	0	0	1	0	0	0
OCMIC (classes)	1	0	0	0	1	0	0	0
OCMEC (instances)	0	0	1	0	0	0	1	0
OCMEC (classes)	0	0	1	0	0	0	1	0
ACAIC (instances)	2	0	0	0	0	0	0	0
ACAIC (classes)	2	0	0	0	0	0	0	0
DCAEC (instances)	0	1	0	1	0	0	0	0
DCAEC (classes)	0	1	0	1	0	0	0	0
ACMIC (instances)	0	1	0	0	0	1	0	0
ACMIC (classes)	0	1	0	0	0	1	0	0
DCMEC (instances)	0	0	0	1	0	0	0	0
DCMEC (classes)	0	0	0	1	0	0	0	0
Number of attributes	2	2	3	0	0	0	0	0
Number of methods	1	1	1	1	1	1	1	1
DIT	2	1	0	0	2	1	0	0
LOC	11	10	11	7	11	10	11	7
	all attributes and methods				**only public attributes and methods**			

Table 2: Metrics as calculated for the sample code from Figure 1.

Note that the second part of the table shows metrics if only public attributes and methods are taken into account. We can see that there is less coupling, because all the attributes in the sample code in Figure 1 are private, so the coupling is related only to methods (which are all public in this case).

1.2 The ES2 Environment

The ES2 metrics analyzer has been implemented on top of the Source Navigator (SN) IDE. SN is a very powerful IDE by itself, especially for the purpose of going through large amounts of code trying to find cross-references and links amongst classes.

We will take a few moments here to explain the choice of using SN, and its advantages and disadvantages. SN is a product made available by RedHat under the GNU General Public License. You can download and install it right away from <http://sources.redhat.com/sourcenav/>. It has been around for a number of years and therefore it is stable and has a relatively large user community. SN extracts quite a number of relevant elements from the source code and stores them in databases, which we then use to compute the metrics. This is ideal since, in principle, for any object-oriented language, the same analyzer can compute the metrics from the database. Furthermore, eventually the ES2 system and its successors will be integrated directly into SN so that metrics will be available as a user navigates through the code.

If you wish to use SN by itself (i.e., you are not interested in the ES2 system), you can download SN directly from the address above. However, if you wish to use ES2, you must use the version of SN bundled in the ES2 distribution as we have made some changes to SN (see section 2 for downloading instructions). ES2 is based on SN version 4.52.

Currently, there is an SN verison 5.0 (just released a few weeks ago!). We have not completed evaluation of SN 5.0 with ES2. Therefore, we reiterate that you must use the SN system that is bundled with ES2.

2 Downloading and Installing ES2

The current version of the ES2 tool works only on **Linux RedHat 7.0 and 7.1** [2]. We do not make any claims about its suitability or applicability under any other conditions.

---

[2] Please check the ES2 Web site for porting to other platforms. There is a continuous effort to port ES2 to other platforms.

To download and install ES2 follow these steps:

1. Download the file SN452_ES2_linux.tar.gz from the following location: <http://www.seg.iit.nrc.ca/~elemam> under the "Tools - Metrics" tab.

2. Save it in the directory where you want it installed.

3. Unpack the downloaded file. Type in the console window in the same directory:

```
tar xvfz SN452_ES2_linux.tar.gz
```

This will create the directory SN452_ES2_linux_unpacked and all the subdirectories (Figure 2).

Figure 2: The tar command to show the directories that are created when you unpack the ES2 file.

4. Change directory to SN452_ES2 by typing: `cd SN452_ES2_linux_unpacked`

5. Run INSTALL by typing: `./INSTALL` (see Figure 3).

Figure 3: Running the installation command.

6. The tool installation starts. It asks the user to accept the licence (Figure 4), and after that it prompts the user to accept or change the offered installation directory (Figure 5). Make sure you have write permission for that directory.

Don't choose for installation the directory `SN452_ES2_unpacked`.

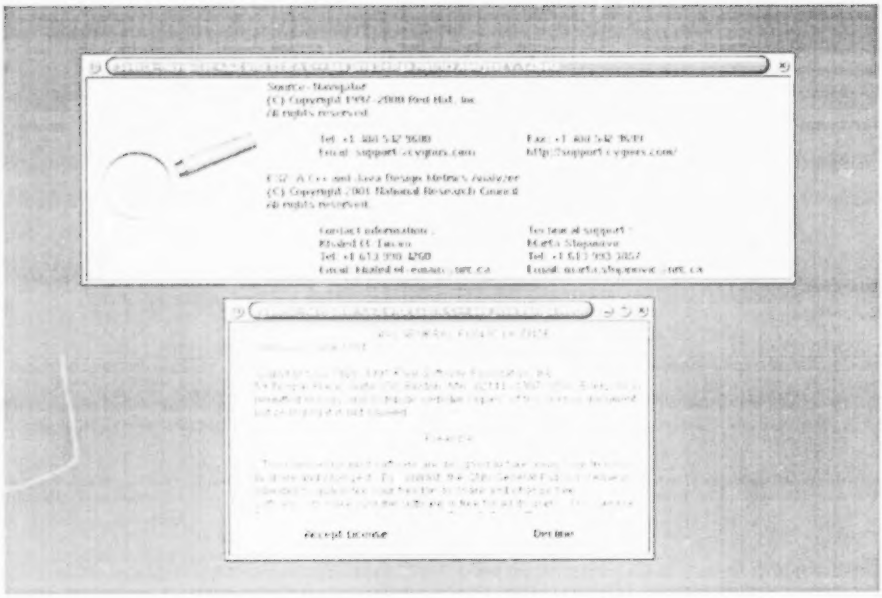

Figure 4: The opening screen when installing ES2.

Figure 5: The window where you will need to provide the installation directory.

7. The installation then continues and asks for the mail tool to use to send bugs. If you are not certain what to put leave it as is; the default is mailhost (screenshot not shown).

8. The installation then prompts for the type of demo projects to install (see Figure 6). These are demo projects for Source Navigator, and it is up to you to make a choice. They are not essential for the ES2 tool.

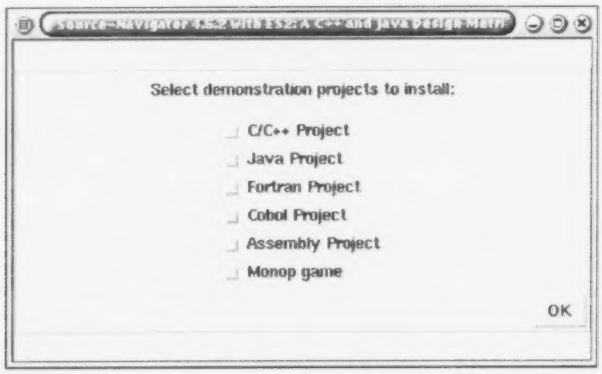

Figure 6: The window where you can select demo projects for SN.

9. The installation proceeds and shows a message to add the installation directory to the environment variable PATH when finished (Figure 7). If you want to have easy access to the tool from any directory, it is useful to set PATH. If you want PATH to take effect only localy to the current terminal window you should set it in the command line (Figure 8). If you want it to take effect in every newly created terminal, you should create or modify the startup file of your shell to add ES2 to your PATH. In order to do this, you should edit the `.bashrc` file[3] found in your home directory (using the Emacs as editor, for instance) by adding the line which sets PATH (see Figure 9).

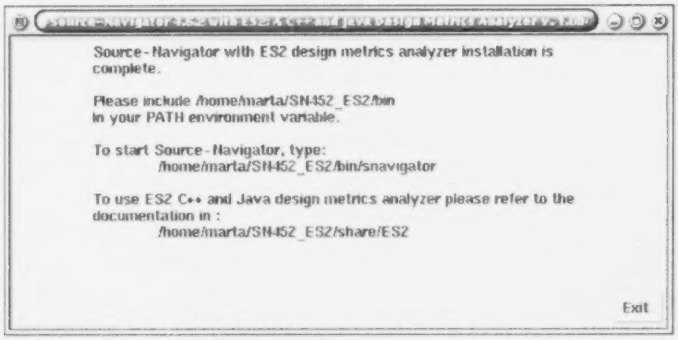
Source-Navigator with ES2 design metrics analyzer installation is complete.
Please include /home/marta/SN-452_ES2/bin in your PATH environment variable.
To start Source-Navigator, type:
/home/marta/SN-452_ES2/bin/snavigator
To use ES2 C++ and Java design metrics analyzer please refer to the documentation in :
/home/marta/SN-452_ES2/share/ES2
Exit

Figure 7: The screen informing the user to modify the PATH environment variable.

[3] The startup file maybe different if you are not using the bash shell.

Figure 8: Setting the PATH environment variable on the command line.

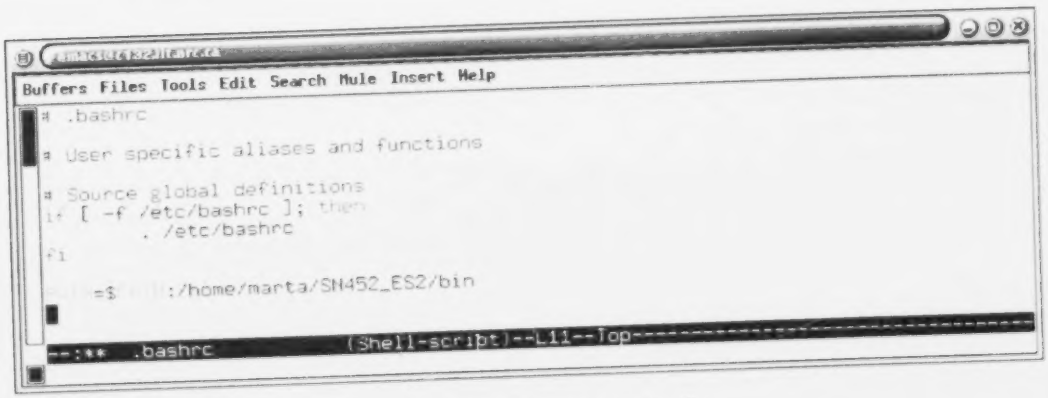

Figure 9: Setting PATH in .bashrc file.

10. Now both Source Navigator 4.52 and ES2 are installed and ready to use. Directory `SN452_ES2/bin` contains, among others, `snavigator` (the script for running Source-Navigator) and `ES2` (the script for running the design metrics analyzer). Figure 8 illustrates how SN is started using the `snavigator` script.

11. N.B. You can remove the directory `SN452_ES2_linux_unpacked` by typing:

```
rm -rf SN452_ES2_linux_unpacked
```

in its parent directory. It is advised to keep the downloaded file (`SN452_ES2_linux.tar.gz`) for eventual future installations.

3 Example of Using ES2

ES2 is a Tcl script (see [12, 15]) that extracts information from database files produced by Source Navigator, calculates design metrics and outputs them in a file easily imported by MS Excel or Sun's StarCalc. It is therefore necessary to first create a Source Navigator project from the source C++ or Java files to be analyzed, and then to run ES2. Note that for C++ only the `.h` files are required, so it does not matter if the `.cpp` files are not available. Here we explain in details how to use both tools.

3.1 Example Systems

The ES2 distribution comes with example systems that can be used to get used to how ES2 works. They can be found in the `SN452_ES2/share/ES2/examples` directory.

C++ examples The subdirectory `SN452_ES2/share/ES2/examples/CPP` contains two C++ example systems. One is a smaller database library called MetaKit (for more information visit <http://equi4.com/metakit/>). It can be found in the subdirectory `SmallExample`. The other one is a more substantial system, a GUI framework called V (<http://objectcentral.com/vgui/vrefman/install.htm>). It can be found in the subdirectory `BigExample`. For both examples we extracted only `*.h` files from the source code since header files are sufficient for our tool (although it also works well with `.cpp` files).

Java examples The subdirectory `SN452_ES2/share/ES2/examples/Java` contains three Java examples: graph, JPublish and J/Top. The Graph class library is a package of Java classes designed to facilitate plotting data using Java applets. It can be downloaded from <http://www.sci.usq.edu.au/staff/leighb/graph/>. Both JPublish and J/Top are created by Anthony Eden and can be downloaded from his Web site: <http://www.anthonyeden.com/>. JPublish is a web application framework which combines the Velocity template engine from the Apache Group with a

content repository and application control framework. J/Top is a Java application that presents data output from 'top' applications on Linux/UNIX machines.

In the next section we show how to use ES2 on the V C++ example. Note that the usage is the same for Java source code.

3.2 Using Source Navigator

As noted earlier, SN is an open-source code navigation tool in which we incorporated our C++ and Java design metrics analyzer. It is invoked from the command line by typing:

```
$home_directory/SN452_ES2/bin/snavigator
```

where `$home_directory` should be replaced with the real home directory of SN452_ES2. If the environment variable PATH is set, it can be invoked just by typing `snavigator` at the prompt.

Here we will explain how to make a Source Navigator project from the sample code included with the ES2 tool. We start with a number of `.h` files from the application called `V`.

1. Upon invoking Source Navigator for the first time it offers the user to create a new project or to find existing ones (Figure 10).

Figure 10: The SN startup screen when there are no projects defined.

2. When the user clicks on the New Project button, a window pops up asking for a project name and source files destination. It offers a default project name and directory (Figure 11), but we will change it. We will call this project `v.proj`, put it in the `/home/marta` directory and use an example from `/SN452_ES2/share/ES2/examples/CPP/BigExample/v` directory (Figure 12).

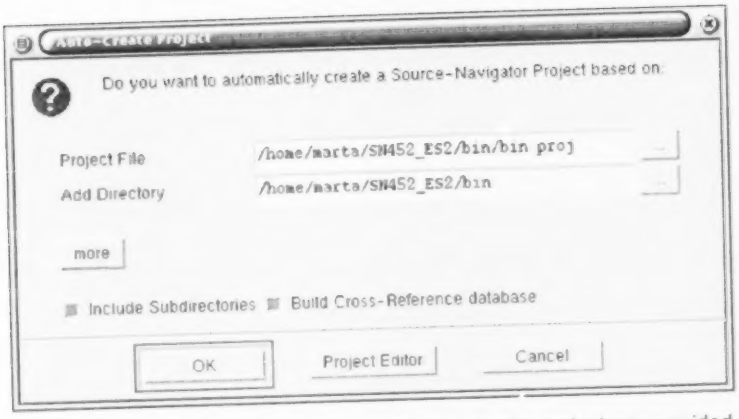

Figure 11: The window where the specific details of a project are provided.

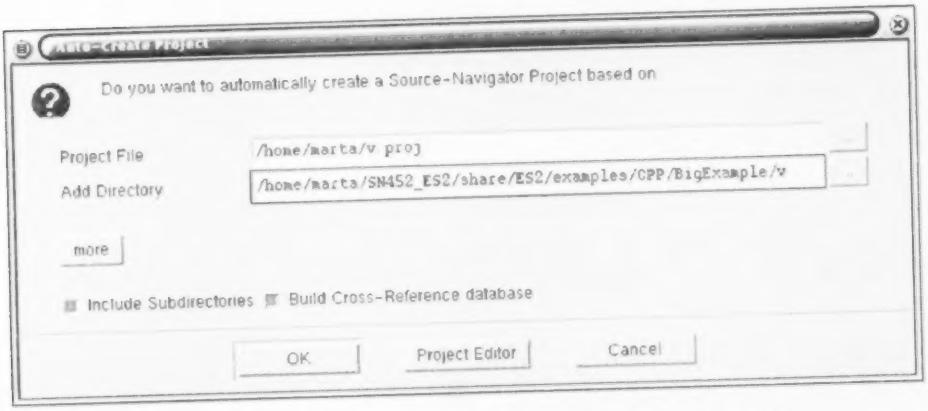

Figure 12: Window when the default project specifics are edited.

Actual directories that should be added to the project for all the included examples are:

For Java:

SN452_ES2/share/ES2/examples/Java/graph or

SN452_ES2/share/ES2/examples/Java/JPublish or

SN452_ES2/share/ES2/examples/Java/JTop.

For C++:

SN452_ES2/share/ES2/examples/CPP/SmallExample/metakit-2.3.4-29 or

SN452_ES2/share/ES2/examples/CPP/BigExample/v

Use these exact directories (and not their subdirectories) in order to preserve the original source directory structure.

3. After clicking on OK, Source Navigator starts parsing the project files and at the end pops up a window with the information on all files (Figure 13).

Figure 13: The SN window with all the files in the project.

4. Now we have created a SN project. By doing so SN has created a number of database files that we use for the purpose of calculating design metrics. Here we can close Source Navigator windows. Of course, it is an excellent code navigation tool and we encourage the user to use it as such. For more information on how to use Source Navigator, see: <http://sources.redhat.com/sourcenav/online-docs/userguide/index_ug.html>.

5. By parsing the source files, SN created the database files that are put by default in $project_home_directory/.snprj directory (Figure 14).

Figure 14: A listing of the database files that are generated by SN.

Some of the database files are used by ES2 to extract all the needed information for calculating metrics. Each file is a table that contains specific symbol information. A list of the database files used, along with a short explanation of their content is given in Table 3.

File Suffix	Table Description	Record Format
cl	Classes	name?start_position?filename;end_position?attributes?{}?{class template}?{}?{comment}
in	Inheritances	class?base-class?start_position?filename;end_position?attributes?{}?{class template}?{inheritance?template}?{comment}
iu	Include	included_file?start_position?include_from_file;0.0?0x0?{}?{}?{}?{}
iv	Instance variables	class?variable-name?start_position?filename;end_position?attributes?{type}?{}?{}?{comment}
ma	Macros	name?start_position?filename;end_position?attributes?{}?{}?{}?{comment}
md	Method definitions	class?name?start_position?filename;end_position?attributes?{ret_type}?{arg_types}?{arg_names}?{comment}
t	Typedefs	name?position?filename;attributes?{original}?{}?{comment}
un	Unions	name?position?filename;attributes?{}?{}?{comment}

Table 3: List of SN database files used as inputs to ES2.

3.3 Using ES2

Now that a Source Navigator project is created, along with the database files, we can invoke ES2. Currently, ES2 works only from the command line.

1. ES2 is invoked in a similar way as Source Navigator, but with arguments. Type:

```
$home_directory/SN452_ES2/bin/ES2 projdir projname access_option
```

where `$home_directory` should be replaced with the real home directory of SN452_ES2.

The command line arguments are:

`projdir`: the directory of the .proj file.

`projname`: project name (without .proj)

`acces_option`: can be `-priv`, `-prot`, `-pub` or `-all`, depending on whether we want to analyze private, protected, public or all variables and methods

`-myhelp`: prints help message (Figure 15).

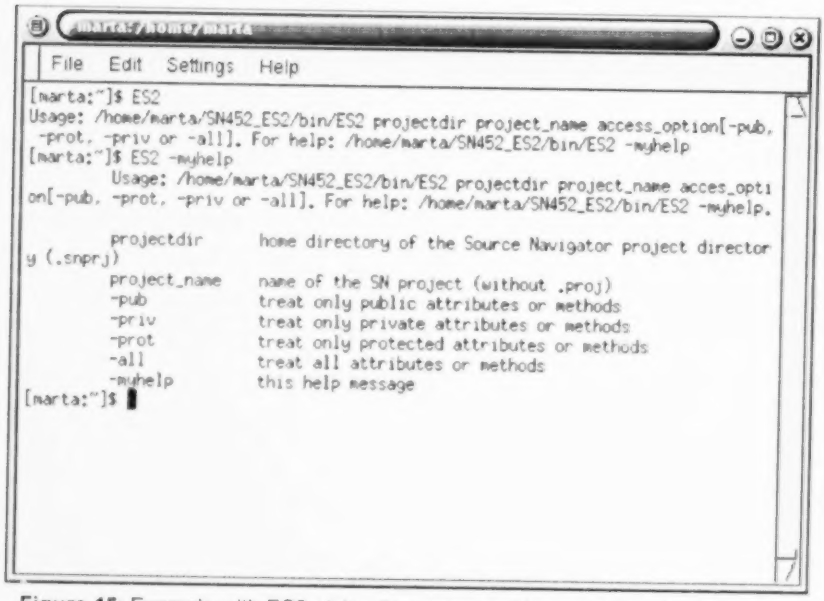
File Edit Settings Help
[marta:~]$ ES2
Usage: /home/marta/SN452_ES2/bin/ES2 projectdir project_name access_option[-pub, -prot, -priv or -all]. For help: /home/marta/SN452_ES2/bin/ES2 -myhelp
[marta:~]$ ES2 -myhelp
Usage: /home/marta/SN452_ES2/bin/ES2 projectdir project_name acces_option[-pub, -prot, -priv or -all]. For help: /home/marta/SN452_ES2/bin/ES2 -myhelp.
projectdir home directory of the Source Navigator project directory (.snprj)
project_name name of the SN project (without .proj)
-pub treat only public attributes or methods
-priv treat only private attributes or methods
-prot treat only protected attributes or methods
-all treat all attributes or methods
-myhelp this help message
[marta:~]$

Figure 15: Example with ES2 giving the usage and help outputs when invoked.

2. For our example (v.proj) we should type (if PATH is set):

```
ES2 /home/marta v -all
```

If PATH is not set it should be:

```
$home_directory/SN452_ES2/bin/ES2 /home/marta v -all
```

Upon pressing ENTER, the metrics extraction starts. Messages are displayed while waiting (Figure 16). For large projects it can take some time. For example, analyzing the V project on a Pentium II 400MHz 128MB RAM takes about 2 minutes.

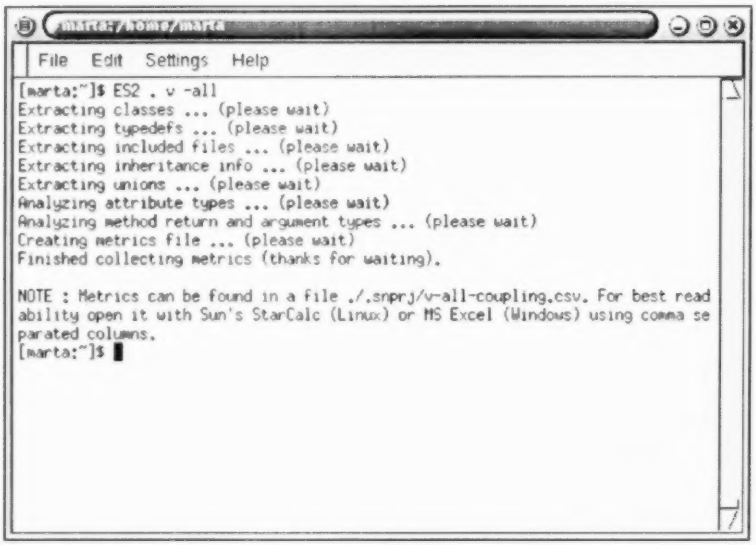
File Edit Settings Help
[marta:~]$ ES2 . v -all
Extracting classes ... (please wait)
Extracting typedefs ... (please wait)
Extracting included files ... (please wait)
Extracting inheritance info ... (please wait)
Extracting unions ... (please wait)
Analyzing attribute types ... (please wait)
Analyzing method return and argument types ... (please wait)
Creating metrics file ... (please wait)
Finished collecting metrics (thanks for waiting).
NOTE : Metrics can be found in a file ./.snprj/v-all-coupling.csv. For best read
ability open it with Sun's StarCalc (Linux) or MS Excel (Windows) using comma se
parated columns.
[marta:~]$

Figure 16: The notification messages provided by ES2 while it is executing.

3. When the execution is finished, two new files will appear in the `./snprj` directory: `v-all-coupling.csv` and `v_coupling.log`. The first file contains calculated metrics, and the second one log messages (like whether a database file was missing, for example). This is illustrated in Figure 17.

Figure 17: After extracting the metrics, two additional files are created.

4. Finally, we should use MS Excel or Sun's StarCalc to visualize the metrics file. Use comma separated columns when importing the file, if it is not imported automatically (Figure 18).

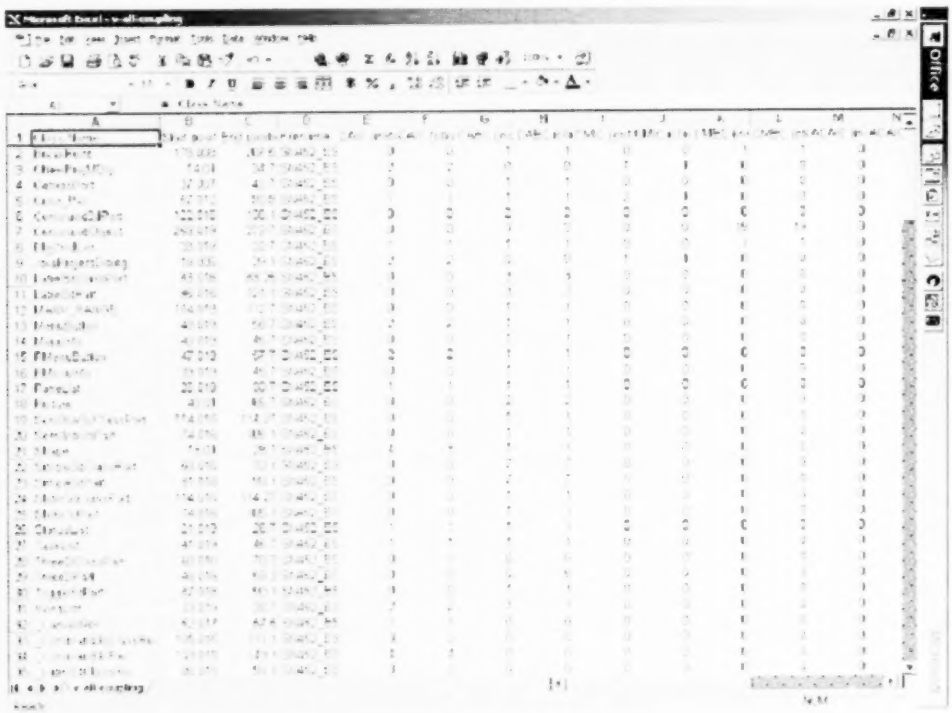

Figure 18: Example of what you should see when opening the .csv file in MS Excel.

Note that the first two columns after the class name give start and end positions of the class: the first number before the dot is the line number, and the second one is the position of the character on that line. The format is taken directly from Source Navigator. It is important to have the position in order to distinguish between classes with the same name (which happens in C++ if you have `#ifdef`).

4 Special Cases

4.1 C++

ES2 works on C++ code, but not all the subtleties of the C++ language are covered. This is partially because Source Navigator's parser does not provide enough information, or because it has been left for future development by us. In this section we will explain the limitations of ES2. In addition, some cases demand more explanations on how the coupling is measured because there were implementation choices that had to be made. Most of the examples are taken from the ACE (Adaptive Communications Environment) library [13, 14].

It should be noted that, based on our experiences extracting metrics from a number of different systems, the limitations of ES2 are minor given that they deal with some cases that do not occur very often (at least in the systems that we have studied thus far). Furthermore, in practice, the extracted metrics are used in statistical analyses. The choices we made (explained below) will usually not have a noticeable influence on the results of the statistical analyses.

4.1.1 Method pointers

Method pointers are not parsed well by Source Navigator. In the case showed in Figure 19, method

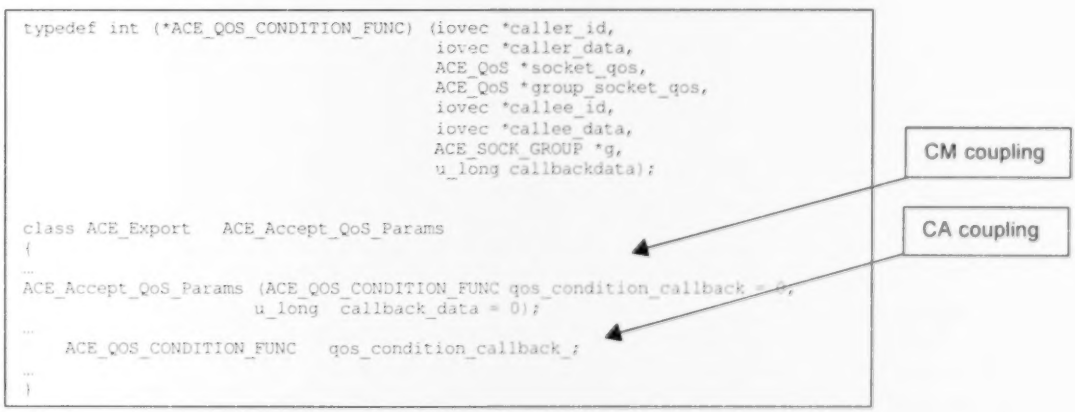

Figure 19: Example of a method pointer to illustrate its impact on computed metrics.

pointer ACE_QOS_CONDITION_FUNC would be presented in the database as int(*)(), i.e., only the return type (`int`) will be present in the database, whereas method argument types will be omitted (`iovec`, `AC_QoS`, `ACE_SOCK_GROUP`, `u_long`). So, we had no choice (without changing the parser) than to limit coupling measurements involving method pointers to their return types.

The question is how to consider coupling with method pointers: as class-attribute or class-method coupling ? We decided to treat it as class-attribute coupling if a method pointer is used as an attribute, and as class-method coupling if it is used as a method's return type, or as a method's argument.

In fact, in the above case, there will be no coupling, because the method pointer's return type is `int`. If the method pointer's arguments were in the database, there would be coupling between `ACE_Accept_QoS_Params` and the method pointer's argument class types (`iovec` and `ACE_QoS`).

4.1.2 Macros

4.1.2.1 Macro definitions

Source Navigator does not pre-process the code, so we did not treat macro definitions even if they can introduce coupling. Figure 20 shows an example of such non-treated coupling. The advantage of not performing any pre-processing is that SN can be very robust compared to other C++ metrics analyzers that we have worked with. For instance, we have been able to start collecting metrics from systems using ES2 within 5 minutes of installation. For other metrics analyzers we sometimes had to spend a few months setting up the environment before metrics could be computed.

```
# if defined (ACE_HAS_TEMPLATE_TYPEDEFS)
...
    # define ACE_MMAP_MEMORY_POOL ACE_MMAP_Memory_Pool
...
# else
...
    # define ACE_MMAP_MEMORY_POOL ACE_MMAP_Memory_Pool, ACE_MMAP_Memory_Pool_Option
...
# endif

class ACE_Export ACE_MEM_SAP
{
    #if (ACE_HAS_POSITION_INDEPENDENT_POINTERS == 1)

       typedef ACE_Malloc_T<ACE_MMAP_MEMORY_POOL, ACE_Process_Mutex, ACE_PI_Control_Block>
MALLOC_TYPE;

    #else

       typedef ACE_Malloc_T<ACE_MMAP_MEMORY_POOL, ACE_Process_Mutex, ACE_Control_Block>
MALLOC_TYPE;

    #endif
...
    MALLOC_TYPE *shm_malloc_;
}
```

Figure 20: Example of macro definitions.

In this case our tool will find class-attribute coupling between ACE_MEM_SAP and ACE_Process_Mutex, ACE_PI_Control_Block and ACE_PI_Control_Block, but not with ACE_MMAP_Memory_Pool and ACE_MMAP_Memory_Pool_Option (macro ACE_MMAP_MEMORY_POOL is not treated).

4.1.2.2 Macro if-else statements

Macro if-else statements pose a different problem. If definitions are protected by macro statements, there can be more than one typedef or class with the same name in the same file. Preprocessing would certainly remove this ambiguity, but when trying to use unprocessed code, one would have to check both definitions. For this reason, we treat them as distinct couplings. Figure 21 shows such a case. Here our tool would find class-attribute coupling between `ACE_ATM_Acceptor` and both `ACE_SOCK_Acceptor` and `ACE_TLI_Acceptor`. Furthermore, from a cognitive perspective (see [7] for a detailed cognitive model justifying the object-oriented computed by ES2), someone comprehending the code will likely trace through the links to other classes for all parts of a guarded `#ifdef`.

```
#if defined (ACE_HAS_FORE_ATM_WS2)

    #include "SOCK_Acceptor.h"
    typedef ACE_SOCK_Acceptor ATM_Acceptor;

#elif defined (ACE_HAS_FORE_ATM_XTI)

    #include "TLI_Acceptor.h"
    typedef ACE_TLI_Acceptor ATM_Acceptor;

#endif

...

class ACE_Export ACE_ATM_Acceptor
{
...
```

Figure 21: Example of macro if-else statements.

4.1.3 Templates

Template classes are treated as any other class for import and export coupling between themselves and other classes. But, template arguments pose a problem: a template class can have an attribute (or method argument or return type) whose type is of the template argument's type. If that template argument is of class type, there should be coupling between these two classes. The problem is that we cannot know the exact type of the template argument until that template class is used, and template arguments are set with real types. So, in order to measure this kind of coupling, we have to find every single usage

of that particular class (in attributes or in methods) and to associate template arguments with all the types they can possibly have. Figure 22 illustrates this case.

```
template <ACE_MEM_POOL_1, class ACE_LOCK>
class ACE_Local_Name_Space: public ACE_Name_Space
{

...

typedef ACE_Allocator_Adapter <ACE_Malloc <ACE_MEM_POOL_2, ACE_LOCK> >
ALLOCATOR;

...

ACE_Name_Space_Map <ALLOCATOR> *name_space_map_;

...
}


template <class ALLOCATOR>
class ACE_Name_Space_Map: public MAP_MANAGER
{

public:
ACE_Name_Space_Map (ALLOCATOR *alloc);

...

}
```

Figure 22: Example of template class usage.

As we can see in Figure 22, class `ACE_Name_Space_Map` has a method argument of type `ALLOCATOR`, which is its template argument type. To calculate coupling we have to know the exact type of `ALLOCATOR`. Therefore, we have to look for uses of that class. One of them is in the class `ACE_Local_Name_Space` that has an attribute of the type `ACE_Name_Space_Map <ALLOCATOR>`, where `ALLOCATOR` is in fact `ACE_Allocator_Adapter <ACE_Malloc <ACE_MEM_POOL_2, ACE_LOCK> >`. Now we can say that the class `ACE_Name_Space_Map` is coupled to `ACE_Allocator_Adapter` and `ACE_Malloc`, as well as with types of `ACE_MEM_POOL_2` (macro, not treated) and `ACE_LOCK` (template argument, whose type will be found in a similar way).

It is obvious that we can only consider coupling between template classes and template arguments after the whole code is processed and every single usage of each template class is found. Template arguments can be of template class' parent type: we check for it when we find that there is coupling, in order to treat it as ancestor-descendant coupling.

Template structures are not parsed in Source Navigator and therefore, are not considered by our tool. In the following case:

```
template <class T>
struct conj_func
{
  T x;
}
```

The ES2 tool will not see coupling between `conj_func` and template argument `T`, but the struct itself will be considered for coupling (i.e., only the template arguments will not be).

4.1.4 Forward declarations

If a class is not in the scope of our class, it can still be used for attribute and method types using forward declaration:

```
class A
{
  class B;
  B b;
}
```

We will consider class-attribute coupling between class A and class B, if class B is defined somewhere in the code, i.e., if it can be found in the class database. If not (which can be the case, if B is supposed to be defined in some `.cpp` file), we do not consider this as a coupling. The reason is simple: we do not have any other way to prove that B is a class other than to look for it in the class database.

In some cases the system is a library and the declared classes will be defined in applications that use the library. For such cases, it is not possible to account for any coupling.

4.1.5 Inner classes

As metrics (such as defects or effort) are usually not calculated for inner classes, we do not consider inner classes in our coupling measurements. But as the parser does not distinguish inner classes from others, we have to explicitly check for them. Figure 23 shows an example of an inner class and its interaction with other classes. If an inner class has an attribute or method type of some other class, we consider that there is coupling between its outmost class and that class (in our example class A is coupled with class C, because of the attribute c in the inner class B). If an outer class has an attribute or method type of an inner class, there is no coupling between them (no coupling between class A and class B because of the attribute b).

```
class A
{
   class B;

   B b;

   class B
   {
     A a;
     C c;
   }
}
```

Figure 23: Example of an inner class.

If a class is coupled to an inner class of some other class, only coupling with the outer class is considered (class C is coupled only with the class A, and not with the inner class B, because of the attribute ab).

4.1.6 Parent classes

The simplest case of ancestor-descendant coupling would be if an attribute or method type is of the parent's class. But there is another case which is also considered as ancestor-descendant coupling. If, for instance, an attribute or method type is a `typedef` defined in a parent class, we consider it as ancestor-descendants coupling even if the `typedef` turns out to be a primitive type. The reason is that in such a case, when trying to understand the code, we will have to look into the parent class to resolve the typedef.

4.1.7 Complex types

We call complex types of the form X::Y. X can be either a class or a typedef (in which case its redefined type is found). Y can be either an inner class (in which case we consider only coupling with X and not Y) or a `typedef` defined in the class X or its parent class, in which case an effort is made to find a type redefined by that `typedef` (if it is of a class type, we consider it as coupling).

In the case of templates we could have the following case:

```
template <class A>
class B
{
  A::C x;
}
```

In this case we consider coupling between class B and a class that can be a type of the template argument A. But we do not go further: we do not test if that class has a field of the name C (inner class or `typedef`). Eventually, it could be done, but as we did not encounter such cases in a number of large systems that we analyzed, we left it unconsidered.

4.1.8 Unions

Unions are not considered as base classes, but we treat them as such and consider that there is coupling between a union and its elements if they are of a class type (as we do for structures).

4.1.9 Operator overloading

In the case of operator overloading the Source Navigator's C++ parser does not parse correctly the return types in the following three situations (in each of these situations the return type will be `int`).

- when the return type is a class type returned by value (if it is returned by reference, it parses it correctly);
- when the return type is a template argument type;
- when the return type is a `typedef`.

Hereafter we show two examples of wrongly calculated coupling caused by this problem. In the following example, instead of `c4_Row` the database file shows `int` (coupling between `c4StringProp` and `c4_Row` is therefore not taken into account):

```
class c4_StringProp: public c4_Property
{
  public:
    ...
    c4_Row operator[] (const char*) const;
    ...
};
```

Or, in this example instead of template argument `TYPE`, the database shows `int` (coupling between a class that could be `TYPE` and `ACE_Atomic_Op` not counted):

```
template <class ACE_LOCK, class TYPE>
class ACE Atomic_Op
{
  public:
    ...

    TYPE operator-- (int);
    ...
}
```

4.1.10 Namespace keyword

Source Navigator does not support the C++ keyword `namespace`. This leads to scoping problems which cannot be resolved if `namespace` support is not added. Right now, the scope of a class is the file in which that class is defined and all the files included by that file. The `namespace` keyword does not require file inclusion: a type defined in the same namespace can be in a file not included by our class' file, and still be visible by our class. If we decide not to define the scope as we did, and to consider all project files, we can have an error in scope when namespace is not used. So, for now, until the C++ parser is changed, it is not advised to use this tool for C++ code that makes extensive use of namespaces.

Even when namespaces are in the same file, classes are not seen by SN if they are declared out of the namespace (in the example below, class A is declared as N::A, because it is part of namespace N). In this case SN will find only class B, and not class A, which again can lead to erroneous metrics.

```
namespace N
{
   class A;
}

class N::A
{
};

namespace M
{
   class B
   {
   };
}
```

4.1.11 Other keywords

Keyword `typename` is not supported by Source Navigator, and our tool simply ignores it (like it ignores keywords `virtual` and `inline` and primitive types). It treats everything as possible types, so this should not be an issue.

4.2 Java

The Java parser in Source Navigator is considerably younger than the C++ parser. Therefore it tends to have more limitations.

4.2.1 Java version

Source Navigator parses Java 1.0. For changes in the Java grammar from Java 1.0 to Java 1.1 and 1.2 see Java grammars from the following site: <http://www.cs.princeton.edu/~appel/modern/java/CUP/>.

4.2.2 Types of local and instance variables

Source Navigator's Java parser is not fully functional: it does not recognize instance and local variable types[4]. It recognizes, however, method parameters and return type. This means that we are able to extract information about class-method coupling (see section 1.1), but not about class-attribute coupling. In the output file the values for class-attribute couplings are therefore all set to 0.

4.2.3 Inner classes

Inner classes in Java are treated the same way as in C++ (see section 4.1.5).

4.2.4 JAR files

Source Navigator does not recognize `.jar` files, so any class in a `.jar` file will not be considered as a part of the system. Jar files are usually used for `.class` files. If that is the case even if they are not zipped, they would not be considered, because it is not source code. If you want those classes to be considered as a part of the system, you should leave them unzipped and in `.java` format, to allow the Java parser to recognize them.

4.2.5 Interfaces

Interfaces are considered as classes: if a class has a method return or argument type of an interface type, it will be considered as coupling. Interfaces can be inherited, so DIT (depth of inheritance tree) is calculated for them also. In the output file no distinction is made between classes and interfaces.

[4] This can be visualised in the Source Navigator's Symbols window: if we choose to see only instance variables, we will see that the column for their type is empty (the same goes for local variables).

4.2.6 Packages

Java core classes are organized in packages. User defined classes can be organized in user defined packages, or, if not, they are put in a default package. Classes are distinguished by their package, which is included in their fully qualified name. For instance, class `String`'s fully qualified name is in fact `java.lang.String` (`java.lang` package is imported by default). We can use its short name, `String`, if there is no other class with the same short name in other imported packages. But if there is such a case, fully qualified names should be used to distinguish between them.

Source Navigator, unfortunately, does not recognize packages. Therefore, it does not recognize fully qualified names, nor does it recognize which class of two classes with the same short name is part of our package or if its in an imported package. This leads to some limitations of our tool, which are presented below.

4.2.6.1 Classes with the same short name

Due to the above-mentioned limitations of Source Navigator's Java parser, if a project contains more classes or interfaces with the same short name, it is not possible to distinguish which one of them is coupled to a class that contains such a type.

For example, in Figure 24 we see that the class `AbstractList`, which implements interface `List` (found in the same package, `java.util`), has a method subList whose return type is of type `List`. The project contains two classes with the name List, but in different packages: the already mentioned interface `java.util.List` and a class `java.awt.List`. As the package `java.awt` is not imported by this file, there is no ambiguity for the compiler: here `List` is in fact `java.util.List`. But, as SN does not distinguish between packages, we do not have enough information to know the exact type of `List`. Therefore, we have no choice but to take into account both of them. In this case, class `AbstractList` will have OCMIC 1 larger than expected, and `java.awt.List` will have OCMEC 1 larger than expected.

```
package java.util;

public abstract class AbstractList extends AbstractCollection implements
List
{

public List subList(int fromIndex, int toIndex)
  {
    ...

    return new SubList(this, fromIndex, toIndex);
  }
}

class SubList extends AbstractList
{
 ...
}
```

```
package java.util;

public interface List extends Collection
{
  ...
}
```

```
package java.awt;

public class List extends Component implements ItemSelectable, Serializable
{
  ...
}
```

Figure 24: Example of a class and an interface with the same name.

4.2.6.2 *Fully qualified class names*

As we mentioned, Java uses fully qualified names for classes (even if programmers usually use short names, for convenience). Unfortunately, Source Navigator puts only short names in the database files, which means that long names are not recognized as class types. So if there is a return, parameter or parent type written as a fully qualified name, the coupling will not be seen. There is probably a way to handle that without changing the parser, but it is left for the next release.

4.2.7 Exceptions and coupling

When we consider coupling between a class and a method we consider method return and parameter types. However, in Java, exception handling can be done by adding a "throws" clause after a method signature. This means that, by parsing a method signature, we can find which exception the class throws. If this exception is our project's class, we could consider coupling between the class that contains that method and the exception class. It would be a special kind of coupling between classes and exception classes. Of course, all exception handling would not be covered by parsing only the method signature (`try-catch` clause is in a method's body), so if we would like to cover it all there should be a major

change in the Java parser. This type of coupling could possibly be measured for C++ code too, as C++ also has a `try-catch` clause.

4.2.8 File extensions

Source Navigator will automatically recognize only `.java` file extensions (and not `.JAVA`, `.jav` or similar), but this can be easily configured in *File-Project Preferences* window of Source Navigator.

5 Software License

This software is copyright © 2000-2001 by the National Research Council of Canada. It is published under the GNU General Public License, which is reproduced in the distribution in a file called "GNUGPL.txt", for the benefit of the software engineering community. This software comes with absolutely no warranty, but we would appreciate bug reports and we will endeavor to fix them.

6 References

[1] S. Chidamber and C. Kemerer, "A Metrics Suite for Object-Oriented Design". In *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493, 1994.

[2] K. El Emam, "Object-Oriented Metrics: A Review of Theory and Practice," in *Advances in Software Engineering: Topics in Comprehension, Evolution, and Evaluation (to appear)*, O. Tanir and H. Erdogmus (eds.): Spinger-Verlag, 2000.

[3] K. El-Emam, S. Benlarbi, N. Goel, and S. Rai, "A Validation of Object-Oriented Metrics," Technical Report, National Research Council of Canada, NRC/ERB 1063 1999.

[4] K. El-Emam and W. Melo, "The Prediction of Faulty Classes Using Object-Oriented Design Metrics," Technical Report, National Research Council of Canada, NRC/ERB 1064 1999.

[5] K. El-Emam, S. Benlarbi, N. Goel, and S. Rai, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics". In *IEEE Transactions on Software Engineering (to appear)*, 2001.

[6] N. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System". In *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797 -814, 2000.

[7] D. Glasberg, K. El Emam, W. Melo, and N. Madhavji, "Validating Object-oriented Design Metrics on a Commercial Java Application," Technical Report, National Research Council of Canada (to appear in the Journal of Systems and Software), NRC/ERB-1080 2000.

[8] W. Harrison, "Using Software Metrics to Allocate Testing Resources". In *Journal of Management Information Systems*, vol. 4, no. 4, pp. 93-105, 1988.

[9] M. Kaaniche and K. Kanoun, "Reliability of a Commercial Telecommunications System". In *Proceedings of the International Symposium on Software Reliability Engineering*, pp. 207-212, 1996.

[10] K.-H. Moller and D. Paulish, "An Empirical Investigation of Software Fault Distribution". In *Proceedings of the First International Software Metrics Symposium*, pp. 82-90, 1993.

[11] N. Ohlsson and H. Alberg, "Predicting Fault-Prone Software Modules in Telephone Switches". In *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886-894, 1996.

[12] J. Ousterhout, *Tcl and the Tk Toolkit*: Addison-Wesley, 1994.

[13] D. Schmidt, "A System of Reusable Design Patterns for Communication Software," in *The Theory and Practice of Object Systems*, S. Berzuk (ed.): Wiley, 1995.

[14] D. Schmidt and P. Stephenson, "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms". In *Proceedings of the 9th European Conference on Object Oriented Programming*, 1995.

[15] B. Welch, *Practical Programming with Tcl and Tk*: Prentice Hall, 1997.